Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Primož Kos

Implementacija alternativnih načinov interakcije s 3D objekti v spletnem brskalniku

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: viš. pred. dr. Marko Privošnik

Ljubljana, 2017

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Implementacija alternativnih načinov interakcije s 3D objekti v spletnem brskalniku:

Interakcija uporabnika grafične aplikacije s 3D objekti običajno poteka z uporabo tipkovnice ali miške, zadnjih nekaj let pa tudi preko zaslona na dotik. Ostale nestandardne naprave, kot je na primer 3D miška, so uporabljene redkeje in s strani operacijskih sistemov niso privzeto podprte. Uporaba tovrstnih nestandardnih naprav za interakcijo s 3D objekti pri uporabi spletnih aplikacij je pogosto še dodatno zapletena, saj namestitev gonilnikov v tem primeru običajno ne zadošča ali pa je taka rešitev pomankljiva. V okviru diplomskega dela preučite možnost interakcije s 3D objekti v okviru spletne WebGL aplikacije z uporabo 3D miške in Leap Motion vmesnika. Implementirajte rešitev, ki omogoča integrirano uporabo obeh nestandardih naprav ter tipkovnice in miške z možnostjo enostavnega preklopa med njimi. Rešitev naj podpira čim več popularnih spletnih brskalnikov in naj bo zasnovana kot namenska vhodno-izhodna knjižnica.

Za usmerjanje pri izdelavi diplomskega dela se zahvaljujem mentorju viš. pred. dr. Markotu Privošniku. Posebna zahvala pa gre družini za potrpljenje in moji punci za vzpodbudo.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Primož Kos sem avtor diplomskega dela z naslovom:

Implementacija alternativnih načinov interakcije s 3D objekti v spletnem brskalniku (angl. *Implementation of alternative ways of interaction with 3D objects in web browser*)

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom viš. pred. dr. Markota Privošnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 5. januarja 2017 Podpis avtorja:

Kazalo

Slike

Povzetek

Naslov: Implementacija alternativnih načinov interakcije s 3D objekti v spletnem brskalniku

V diplomskem delu obravnavamo implementacijo alternativnih načinov interakcije s 3D objekti v spletnem brskalniku. V ta namen smo najprej naredili analizo obstoječih načinov interakcije in njihovih implementacij v spletnem brskalniku. Rezultati analize so predstavljali osnovo, na kateri smo gradili lastno rešitev, ki predstavlja zanesljivo in za uporabnika naravno interakcijo. Izdelali smo namensko vhodno-izhodno knjižnico, ki zna zajemati podatke štirih različnih naprav - miške, tipkovnice, vmesnika Leap Motion in 3D miške, ter jih pravilno pretvoriti v enotne ukaze, ki skrbijo za premik 3D prizora. Knjižnica je namenjena za uporabo v spletnem brskalniku in smo jo zato napisali v jeziku JavaScript. Z uporabo objektnega načina programiranja smo dosegli neodvisnost modulov, kar omogoča preprosto razširitev in dodajanje novih funkcionalnosti brez vpliva na ostale module. 3D prizor izrisujemo s pomočjo knjižnice three.js, ki uporablja JavaScript API vmesnik WebGL.

Ključne besede: Alternativni načini interakcije, Leap Motion, 3D miška, WebGL, three.js.

Abstract

Title: Implementation of alternative ways of interaction with 3D objects in web browser

In this thesis we present implementation of alternative inputs for interactions with 3D objects in a web browser. In order to achieve this purpose we conducted analysis of existing modes of interactions and their implementations in a browser. Results of the analysis served as a foundation on top of which we built our own solution, which presents a robust and user friendly interaction.
We programmed a dedicated input/output library, which is capable of capturing input data from four different input devices: mouse, keyboard, Leap Motion controller and 3D mouse, and transform it into unified commands that control the 3D scene. The library is browser based and is therefore written in JavaScript. Using object oriented way of programming we were able to achieve code module autonomy, which then allows adding new functionality without interference with other modules. The 3D scene is rendered using three.js library, which is based on WebGL JavaScript API.

Keywords: Alternative interaction methods, Leap Motion, 3D mouse, WebGL, three.js.

Poglavje 1

Uvod

Trend razvoja računalniških aplikacij v zadnjih letih nakazuje premik od klasičnih sistemskih aplikacij proti spletnim različicam le-teh. Večje število uporabnikov, preprostost dostopa, dostop do spleta na vsakem koraku in enostavnejši razvoj aplikacije za različne operacijske sisteme in mobilne naprave hkrati so le nekateri od ključnih razlogov, zaradi katerih se vse več podjetij in razvijalcev odloči, da svojo storitev ali aplikacijo ponudijo uporabnikom prek spleta.

Podobni trendi so opazni tudi v 3D grafiki, ki je bila nekoč izključno domena sistemskih aplikacij na zmogljivih računalnikih, danes pa je prisotna že v spletnem brskalniku na našem telefonu. Poleg spletnih iger se 3D grafike v spletnem brskalniku poslužujejo tudi oblikovalci in umetniki, arhitekti, medicinska industrija, digitalne agencije za oglaševalske kampanje, avtomobilska industrija za vizualne konfiguratorje avtomobilov in podobne sfere, ki uporabljajo spletno grafiko.

Vzporedno s porastom osebnih računalnikov, mobilnih naprav in dostopnostjo do spleta se pojavljajo tudi novi načini interakcije z računalniki in ostalimi napravami, kot na primer "dotik"(angl. touch) in bolj nišni načini interakcije, kot na primer digitalne grafične tablice, različni senzorji premikanja (npr. Microsoftov Kinect), 3D svinčniki in miške, EEG naprave, ki zaznavajo električno aktivnost možganov in še mnogi drugi.

V tem diplomskem delu bomo za interakcijo s 3D objektom poleg miške in tipkovnice implementirali tudi dva alternativna vmesnika: Leap Motion, ki zaznava premikanje rok s pomočjo infrardečih kamer in LED sijalk, ter 3D miško 3Dconnexion SpaceNavigator.

Glavni cilj dela je izdelati JavaScript knjižnico, ki bo uporabniku omogočala upravljanje 3D prizora v spletnem brskalniku s pomočjo prej naštetih štirih vmesnikov. Knjižnica mora znati komunicirati z Javascript vmesnikom *WebGL API*, ki se uporablja za izris 3D grafike v spletnem brskalniku. Na spletu nismo našli nobene knjižnice, ki bi obsegala podobne funkcionalnosti.

Stremeli smo k temu, da bi bila naša knjižnica fleksibilna in preprosta za implementacijo. Fleksibilnost smo dosegli z modularnostjo kode (opisano v poglavju 3.1), kar omogoča uporabniku razširitev funkcionalnosti z dodajanjem novih modulov (na primer za druge vmesnike) ali spreminjanje obstoječih (implementacija kakšne druge knjižnice za izris 3D prizora). Vsi moduli razširjajo glavni prototip knjižnice in tako lahko tudi dedujejo podatke. Tak način strukture se imenuje prototipni vzorec (angl. *Prototype pattern*) in ga v različnih oblikah uporablja večina popularnih modularnih knjižnic, na primer jQuery, Dojo in podobne. V našem primeru ima vsak modul svojo inicializacijsko funkcijo, s katero se vključi njegova funkcionalnost, zato je za inicializacijo naše celotne knjižnice potrebnih sedem vrstic JavaScript kode (opisano v poglavju 3.4). Knjižnico smo razvijali in testirali v spletnem brskalniku Google Chrome, vendar knjižnica podpira tudi uporabo v spletnih brskalnikih Mozilla Firefox in Safari 11.

Za izris slike smo uporabili JavaScript knjižnico *three.js*, ki poenostavi uporabo vmesnika *WebGL API*. Za prikaz delovanja vmesnikov smo pripravili 3D prizor, ki vsebuje 500 piramid, naključno razporejenih po prostoru, v katerem premikamo kamero. Izris omogoča eden izmed modulov (opisan v poglavju 5), implementirali pa smo tudi modul, ki uporabniku prikaže statusno vrstico na dnu okna in prikazuje, kateri vmesnik je trenutno aktiven ter v katerem načinu upravlja s kamero (opisan v poglavju 3.3).

Diplomsko delo je razdeljeno v štiri poglavja - v prvem bomo opisali vsako

od vhodnih naprav in orodja, ki smo jih uporabili za izdelavo knjižnice. V drugem poglavju bomo pojasnili, kako je naša knjižnica zgrajena in kako si posamezni moduli izmenjujejo podatke ter kako jih inicializiramo. Tretje poglavje opisuje vsak vmesnik posebej, način zajemanja podatkov ter pretvorbo le-teh v ukaze za premik kamere. V zadnjem poglavju bomo opisali izris 3D prizora in izračun pozicije kamere v prostoru na podlagi prejetih podatkov.

Poglavje 2

Vhodne naprave in orodja

V tem poglavju bomo predstavili metode programiranja, orodja in programske jezike, ki smo jih uporabili ter opisali vse štiri vmesnike, ki smo jih implementirali v knjižnici, njihovo zgodovino in način delovanja.

2.1 Vhodne naprave

V naši knjižnici smo implementirali podporo za štiri različne vhodne naprave: miško, tipkovnico, 3D miško in Leap Motion vmesnik.

2.1.1 Miška

Računalniška miška je vhodna naprava, ki zaznava dvodimenzionalne premike relativno na njeno podlago. Tipično se ti premiki pretvorijo v premik kurzorja na zaslonu, v našem primeru pa premikajo kamero relativno na prizor.

Sledilno kroglico (angl. *Trackball*), predhodnico miške, je izumil Ralph Benjamin za namene britanskega radarskega omrežja v času 2. svetovne vojne. Leta 1965 pa so jo inženirji v Nemčiji obrnili na glavo in izumili napravo, ki je delovala po principu današnjih mišk. Danes poznamo več vrst mišk, ki se delijo v dve glavni skupini - mehanske, ki uporabljajo sledilno kroglico, in optične, ki za sledenje tipično uporabljajo infrardečo svetlobo ali laser. Poleg kroglice ali optičnega senzorja miške danes tipično sestavljata še

vsaj dva gumba za levi in desni klik ter kolesce, s katerim se premikamo gor ali dol po dokumentih, straneh, seznamih itd.

2.1.2 Tipkovnica

Tipkovnica je tipično namenjena vnosu črk, številk in drugih znakov v urejevalnik besedila ali za interakcijo z operacijskim sistemom. Sestavlja jo večje število tipk, ki so razporejene po določenem sistemu, ki se lahko razlikuje glede na to, kateri jezik ali operacijski sistem uporabljamo.

Tipkovnice so se razvile iz pisalnih strojev, ki so se uporabljali več kot 150 let preden so jih nadomestili računalniki. Kljub prehodu na računalnike se je na tipkovnicah obdržala razporeditev tipk, ki se je uporabljala na pisalnih strojih in je bila zasnovana tako, da se ob hitrem tipkanju kladivca pogosto uporabljenih črk med sabo niso zatikala.

2.1.3 3D miška 3Dconnexion SpaceNavigator

3D miške so se začele pojavljati v devetdesetih letih prejšnjega stoletja s porastom računalniške 3D grafike. Najbolj znane so miške podjetij Logitech in 3Dconnexion. Namenjene so predvsem za 3D modeliranje, arhitekturo in okolja, kjer je pomembno natačno premikanje skozi 3D prostor ali interakcija in manipulacija 3D objekta.

Za razvoj naše knjižnice smo uporabili miško SpaceNavigator nemškega podjetja 3Dconnexion. Miška je sposobna zaznavati premike v šestih različnih smereh (tehnologija poimenovana 6DOF - six degrees of freedom). Za premikanje uporabljamo velik črn analogni krmilnik, ki je po obliki podoben pokrovčkom na plastenkah ali manjši gobi in je prevlečen z gumo za boljši oprijem. Ogrodje miške ima ob straneh tudi dva gumba, ki jim lahko sami določimo funkcije. Ogrodje je narejeno iz težke kovine, kar naredi miško lažjo za upravljanje, saj se ne premika ob potiskih in potegih. Z računalnikom se poveže preko USB vmesnika in za delovanje potrebuje namestitev dodatnih gonilnikov.

Slika 2.1: 3D miška SpaceNavigator in različni načini upravljanja [9]

2.1.4 Leap Motion

Leap Motion vmesnik je majhna USB vhodna naprava, ki je zmožna zaznavati premike rok in prstov v 3D prostoru. S pomočjo dveh monokromatskih infrardečih kamer in treh infrardečih svetlečih diod (angl. *light-emitting diode - LED*) je v polokroglem prostoru do razdalje enega metra naprava zmožna zaznavati premike z natančnostjo do 0,7 milimetra. Diode oddajajo enakomerno svetlobo in kameri zajemata do 200 slik na sekundo. Ker sta kameri druga od druge oddaljeni približno 3 centimetre, je s primerjavo dveh slik priložena programska oprema zmožna matematično izračunati pozicijo objekta ali točke v 3D prostoru. Za delovanje vmesnika mora uporabnik naložiti ustrezne gonilnike in programsko opremo, dosegljivo na spletnem

mestu proizvajalca.

Slika 2.2: Upravljanje z Leap Motion vmesnikom [10]

2.2 JavaScript

Naša knjižnica je napisana v jeziku JavaScript. Uporabljali smo tudi novo ECMAScript 2015 sintakso, ki jo bomo podrobneje opisali v nadaljevanju.

2.2.1 Opis in zgodovina

JavaScript je visoko nivojski dinamični programski jezik, uporabljen predvsem za razvoj interaktivnih spletnih aplikacij na strani odjemalca. V za-

dnjih letih pa mu raste priljubljenost tudi za izvajanje kode na strežniku, predvsem z razvojem izvajalnega okolja NodeJS. JavaScript je standardiziran na podlagi ECMAScript specifikacije.

JavaScript je leta 1995 začel razvijati Brendan Eich za spletni brskalnik Netscape Navigator 2, kjer je bil njegov primarni namen preverjanje vnešenih podatkov v formah. V času telefonskih modemov so bile zahteve poslane na strežnik drage, zato se je pojavila potreba po preverjanju podatkov na strani uporabnika brez pošiljanja na strežnik.

2.2.2 ECMAScript 2015

V diplomskem delu smo JavaScript kodo pisali v sintaksi ECMAScript 2015 (staro ime ECMAScript 6). To je dolgo pričakovana nadgradnja JavaScripta (prva večja po šestih letih [1]), ki je prinesla obsežnejše izboljšave in nove funkcionalnosti.

V času razvoja nove verzije je bila ta kot logična naslednica verzije ES5 poimenovana ECMAScript 6 (ali ES6 na kratko), vendar so se člani ECMAScript komiteja odločili, da si jezik zasluži pogostejše nadgradnje in spremembe, zato so ES6 razdelili v več manjših nadgradenj, ki jih bodo izdajali vsako leto in primerno tudi preimenovali izdajo v ES2015 (naslednja bo ES2016 itd.). Hkrati so s tem razvijalcem brskalnikov in JavaScript interpreterjem omogočili tudi hitrejšo adaptacijo novih standardov in poenotenju implementacij ter tako tudi zmanjšali potrebo po prevajalnikih sintakse JavaScript (angl. *source-to-source compiler* ali *transpiler*), kot npr. BabelJS.

Nekatere nove funkcionalnosti iz ES2015, uporabljene v tem diplomskem delu, so krajša oblika zapisa funkcij (angl. *arrow functions*), lokalna inicializacija spremenljivk (angl. *let variable*), metode kot lastnosti objekta (angl. *method properties*), literal znotraj niza znakov (angl. *string literals*) in druge.

2.3 Git

Pri razvoju kode za naše diplomsko delo smo uporabljali Git kot sistem za upravljanje z izvorno kodo. Za gostovanje našega projekta smo uporabili Github. Projekt je dostopen na `https://github.com/pr1moz/Interact-io`.

2.3.1 Opis in zgodovina

Sistem Git je razvil Linus Torvalds leta 2005 za potrebe razvoja jedra operacijskega sistema Linux kot distribuiran sistem za upravljanje z verzijami. Njegove glavne prednosti so hitrost, zagotavljanje integritete podatkov in možnost distribuiranega, nelinearnega dela večih razvijalcev hkrati.

2.4 IntelliJ IDEA

Za pisanje kode v našem diplomskem delu smo uporabljali integrirano razvojno okolje (IDE) IntelliJ IDEA razvijalca JetBrains. Namenjeno je predvsem za razvoj Java projektov zaradi odlične podpore za ta jezik, vendar je z uporabo nekaterih vtičnikov primerno tudi za večino drugih jezikov. Funkcionalnosti tega okolja, kot so indeksiranje izvorne kode, preverjanje kakovosti kode, samodejno dopolnjevanje in predlaganje kode ter integracija s sistemom Git, so nam omogočile hitrejše in boljše pisanje kode.

Poglavje 3

Knjižnica

Kot smo zapisali v povzetku, je bil namen našega diplomskega dela ustvariti knjižnico, ki bo omogočala zajem podakov izbranih vnosnih naprav ter jih v ustreznem formatu posredovala delu kode, ki skrbi za izris 3D prizora v spletnem brskalniku. V našem primeru je to knjižnica three.js.

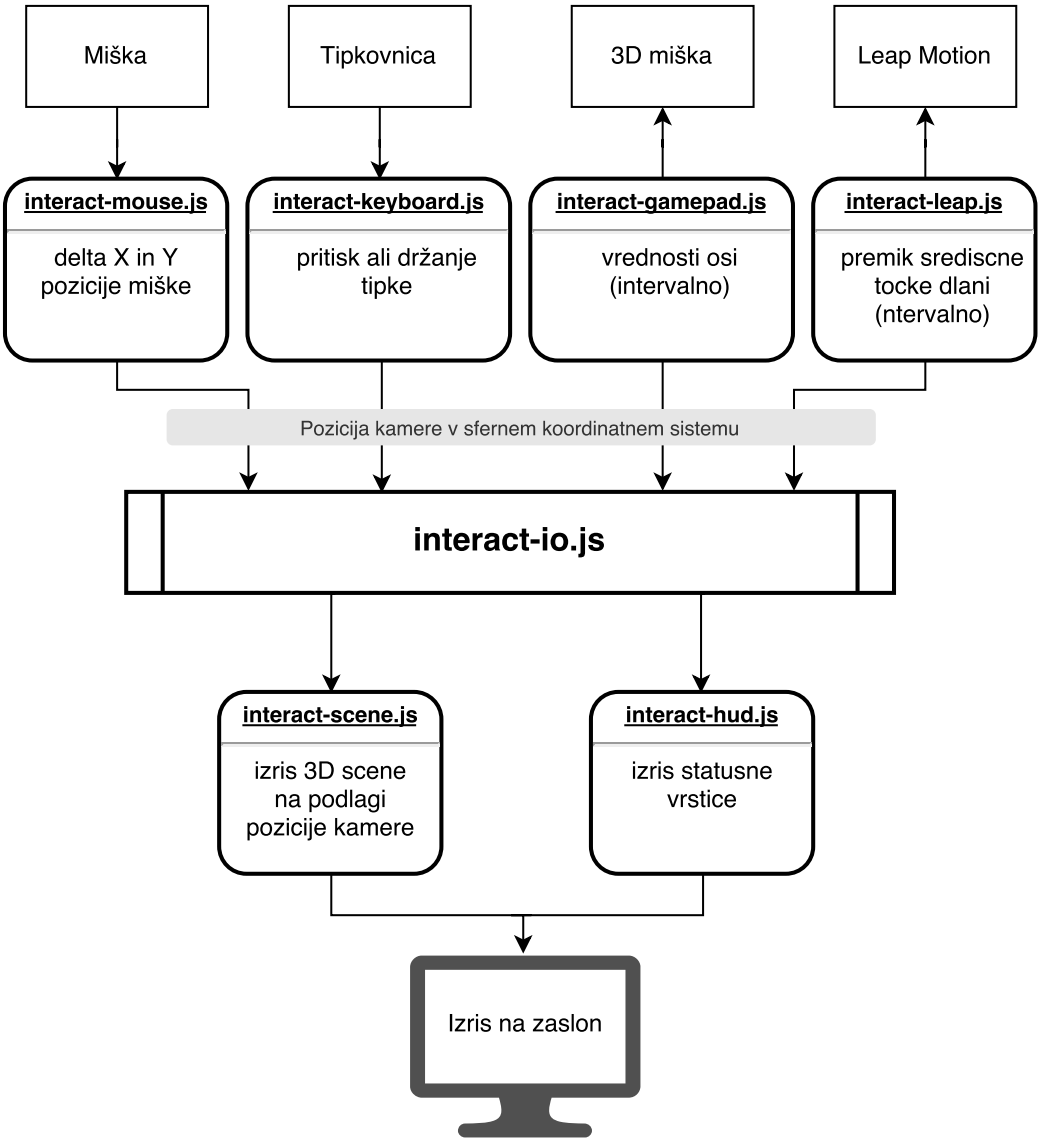
3.1 Zgradba in moduli

V naši knjižnici smo uporabili objektni način programiranja, natančneje z uporabo prototipnega vzorca (angl. *Prototype pattern*), saj nam ta omogoča lažjo uporabo neodvisnih modulov za vsako vnosno napravo ali samostojni smiselni del kode posebej ter deljenje skupnih podatkov med njimi. Prototipni vzorec programiranja temelji na ustvarjanju primerkov vnaprej definiranega funkcijskega objekta. Te objekte lahko smatramo kot predloge ali načrt, kaj bo vsak primerek obsegal in jim pravimo prototipi. Ta način programiranja je zelo podoben uporabi razredov v drugih programskih jezikih, ki pa jih JavaScript nima in zato uporablja prototipe (razredi dodani v verziji ES2015 so zgolj nadgradnja prototipov). Prednost take oblike programiranja je, da uporabljamo prednosti JavaScripta, namesto da posnemamo načine, ki se uporabljajo v Javi in ostalih programskih jezikih. Podpira tudi dedovanje, ki ni le enostavno za uporabo, ampak hkrati tudi nudi hitrejše delovanje, saj

vsak primerek hrani samo referenco na metodo, definirano v prototipu, in ne celotne kopije metode [8].

Knjižnico sestavlja sedem JavaScript datotek:

- *interact-io.js* - glavna datoteka, ki definira ogrodje, v katerega se vsi ostali moduli povežejo, skupno bazo nastavitvenih spremenljivk ter nekaj pomožnih objektnih tipov, ki olajšajo delo z 2D in 3D vektorji,
- *modules/interact-mouse.js* - modul, ki skrbi za zajem podatkov z navadne miške,
- *modules/interact-keyboard.js* - modul, ki posluša pritiske gumbov na tipkovnici,
- *modules/interact-leap.js* - modul, ki skrbi za intervalni zajem podatkov z vmesnika Leap Motion,
- *modules/interact-gamepad.js* - modul, ki skrbi za zajem podatkov 3D miške, poimenovan gamepad, ker uporablja JavaScript API vmesnik Gamepad, sicer namenjen za povezovanje z igralnimi kontrolerji,
- *modules/interact-scene.js* - modul, ki skrbi za izris 3D prizora s pomočjo three.js knjižnice,
- *modules/interact-hud.js* - modul, ki na dnu zaslona izriše informacijski element, ki nakazuje, katera naprava je trenutno v uporabi in katera interakcija se izvaja.



Slika 3.1: Prikaz zgradbe knjižnice in potek podatkov

V nadaljevanju bomo podrobneje opisali glavni modul in HUD modul, ostali moduli pa bodo obrazloženi v ločenih poglavjih.

3.1.1 Vtični moduli

Našo knjižnico smo poimenovali *Interact-io.js*, prototip, iz katerega kloniramo primerek ob inicializaciji, pa smo poimenovali *INTERACT*.

```
1 function INTERACT (cont) {
2   this.nekaLastnost = true;
3   // koda glavnega modula
4 }
```

Izsek kode 3.1: Definicija glavnega modula v interact-io.js

Glavni modul lahko razširimo z dodatnimi vtičniki ali moduli, ki lahko dostopajo do lastnosti glavnega modula, ki ga razširjajo, ne pa tudi do lastnosti ostalih modulov, kar nam omogoča neodvisnost modulov. Vsak modul doda v glavnega svojo funkcijo, s pomočjo katere nato incializiramo ta modul.

```
1 INTERACT.prototype.initMouse = function () {
2   console.log(this.nekaLastnost); // true
3   // koda modula miške
4 }
```

Izsek kode 3.2: Primer razširitve z modulom za miško in dostopanje do lastnosti

3.2 Glavni modul

Ker je večji del funkcionalnosti naše knjižnice razdeljen v več modulov, glavni modul služi samo kot ogrodje za ostale in za hranjenje splošnih nastavitev, ki jih lahko uporabnik po želji tudi spreminja. Nastavitve, ki so na voljo, med drugim vključujejo hitrost rotacije, premikanja in približevanja kamere, prag odzivnosti 3D miške in vmesnika Leap Motion ter njuno občutljivost.

```
function INTERACT (cont) {
  // Interval (v ms) za preverjanje stanja vmesnikov
  this.timerInterval = 30;

  // Modifikatorji interakcij
  this.zoomSpeed = 1.0;
  this.rotateSpeed = 1.0;
  this.panSpeed = 7.0; // v pikslih
}
```

Izsek kode 3.3: Primer različnih nastavitev v glavnem modulu

Poleg nastavitev so v glavnem modulu definirani tudi prototipi, ki so potrebni izris prizora, kot naprimer dvo- in tridimenzionalni vektorji ter sferni koordinatni sistem. Vsak tip ima tudi nekaj lastnih metod, ki olajšajo računanje in transformacijo podatkov iz ene oblike v drugo. Določili smo tudi privzete začetne vrednosti njihovih lastnosti v primeru, da uporabnik ne želi določiti specifične vrednosti ob ustvarjanju novega primerka.

```
INTERACT.Vector2 = function (x, y) {
  this.x = x || 0;
  this.y = y || 0;
  return this;
};
INTERACT.Vector2.prototype = {
  set: function (x, y) {
    this.x = x;
    this.y = y;
    return this;
  },
  getDelta: function (a, b) {
    this.x = a.x - b.x;
    this.y = a.y - b.y;
    return this;
  }
```

```
17 };
```

Izsek kode 3.4: Primer prototipa 2D vektorja in njegovih lastnosti

Prototipe za vektorje in sferni koordinatni sistem smo naredili po zgledu knjižnice *three.js*, ki smo jo uporabili za izris prizora in uporablja še veliko bolj obsežen skupek prototipov in metod za lažje računanje vektorjev.

3.3 HUD

HUD je kratica za angleški izraz Head-up display, ki pojmuje tip transparentnega prikazovalnika, ki uporabniku prikazuje določene podatke, ne da bi mu zastrl pogled na dogajanje za njim. Tak tip uporabniškega vmesnika se velikokrat uporablja v računalniških igricah, kjer igralcu tipično prikazuje stanje njegovega napredka, zdravje, zalogo streliva in podobno. Fizična oblika takega prikazovalnika se na primer uporablja v vojaških lovskih letalih, kjer računalnik s projeciranjem informacij na kos stekla pilotu olajša navigiranje in ciljanje tarč.

V našem primeru pa je HUD temna vrstica na dnu okna brskalnika, ki uporabniku sporoča, katera vhodna naprava trenutno upravlja s kamero in kateri način interakcije se uporablja (rotacija, premik ali približevanje). Stanje je prikazano z dvema skupinama ikonic, na levi za vhodne naprave in na desni za način interakcije. Tako kot ostali moduli je HUD modul opcijski in nujen za delovanje knjižnice in ostalih modulov.

Slika 3.2: HUD prikazovalnik

Pomen ikon je sledeč: (1) - miška, (2) - tipkovnica, (3) - 3D miška, (4) -

vmesnik Leap Motion, (5) - rotacija kamere, (6) - premik kamere, (7) - približevanje/oddaljevanje kamere. Trenutno aktivna akcija ali vhodna naprava je označena s svetlejšo ikono. Na sliki 3.2 sta aktivni miška in rotacija kamere.

Za oblikovanje smo uporabili CSS, ki se nahaja v datoteki */css/hud.css*, ikonice pa smo prenesli s spletnega mesta Flaticon [2] v formatu SVG in jih rahlo priredili našim potrebam.

3.4 Inicializacija

Inicializacija naše knjižnice je zelo preprosta. Predpogoj za inicializacijo glavnega modula je HTML datoteka, ki naloži datoteko *interact-io.js*. Če bomo inicializirali tudi katerega od drugih modulov, moramo v primeru modula za vmesnik Leap Motion najprej naložiti tudi potrebno knjižnico *leap-0.6.4.js*. Za izris prizora pa je potrebna knjižnica *three.js*. Obe zunanji knjižnici se nahajata v mapi *js/lib*.

Za inicializacijo knjižnice moramo v dokumentu HTML določiti tudi en element, v katerega se bo izrisoval prizor. Ta element podamo kot argument ob klicu funkcije z ukazom:

```
window.interact = new INTERACT(document.getElementById('element'));
```

V tem primeru smo v dokumentu poiskali element, ki ima atribut *id* poimenovan "element"in ga poslali v funkcijo. Z ukazom `new` smo naredili nov primerek našega prototipa, definiranega v glavnem modulu in jo shranili v spremenljivko *interact*, ki je del globalnega objekta *window*.

Za tem lahko inicializiramo še ostale module. To storimo tako, da pokličemo njegovo inicializacijsko funkcijo, ki je bila dodana v prototip glavnega modula ob razširitvi in je sedaj del naše spremenljivke. Primer za inicializacijo modula miške je torej sledeči:

```
interact.initMouse();
```

Enako storimo z vsemi moduli, ki jih želimo vključiti. Celoten primer datoteke HTML in inicializacije iz naše knjižnice spodaj.

```
1 <!-- Pripravimo dokument HTML -->
2 <!DOCTYPE html>
3 <html lang="en">
4 <head>
5     <title>Interact-io demo</title>
6     <meta charset="utf-8">
```

```
7       <meta name="viewport" content="width=device-width,
            user-scalable=no, minimum-scale=1.0, maximum-scale=1.0
            ">
8
9       <!-- Naložimo potrebne zunanje knjižnice -->
10      <script src="js/lib/three.js"></script>
11      <script src="js/lib/leap-0.6.4.js"></script>
12
13      <!-- Naložimo glavni modul -->
14      <script src="js/src/interact-io.js"></script>
15
16      <!-- Naložimo module za posamezno napravo -->
17      <script src="js/src/modules/interact-mouse.js"></script>
18      <script src="js/src/modules/interact-keyboard.js">
            </script>
19      <script src="js/src/modules/interact-gamepad.js"></script
            >
20      <script src="js/src/modules/interact-leap.js"></script>
21
22      <!-- Naložimo modul za izris prizora -->
23      <script src="js/src/modules/interact-scene.js"></script>
24
25      <!-- Naložimo HUD modul -->
26      <script src="js/src/modules/interact-hud.js"></script>
27
28      <!-- Nekaj CSS oblikovanja za pripravo izrisa -->
29      <style>
30        html, body {margin: 0; padding: 0; overflow: hidden}
31      </style>
32  </head>
33  <body>
34    <!-- Element v katerega se bo izrisoval prizor -->
35      <div id="container"></div>
36
37    <!-- Blok z inicializacijsko JavaScript kodo -->
38      <script>
39
40        // Ustvarimo nov primerek INTERACT prototipa in jo
```

```
            shranimo v globalni objekt window, da je dosegljiva
            uporabniku
41          window.interact = new INTERACT(document.getElementById(
            'container'));
42
43          // Inicializiramo module naprav
44          interact.initMouse();
45          interact.initKeyboard();
46          interact.initGamepad();
47          interact.initLeap();
48
49          // Inicializiramo modul za izris prizora
50          interact.renderScene();
51
52          // Inicializiramo HUD modul
53          interact.initHUD();
54      </script>
55  </body>
56  </html>
```

Izsek kode 3.5: Primer in razlaga inicializacije knjižnice

Poglavje 4

Zajem podatkov

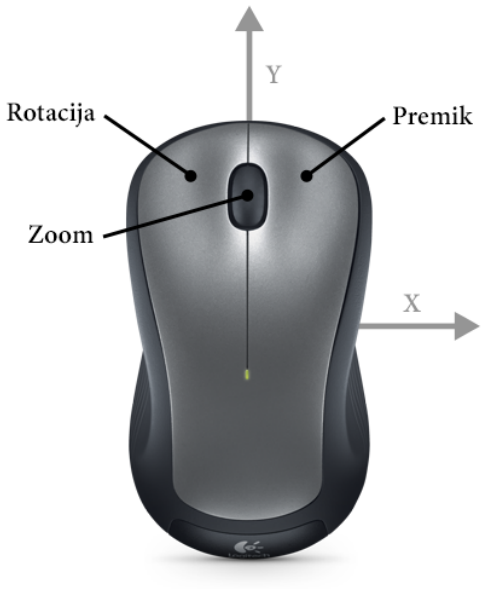
V tem poglavju bomo opisali, kako upravljamo s posamezno napravo ter kako naša knjižnica zajame podatke interakcije za vsako od podprtih vhodnih naprav. Te so med sabo neodvisne in za njihovo delovanje ni potrebno preklapljanje med njimi, nismo pa podprli interakcije z več napravami hkrati, saj bi prihajalo do šumov in delovanje ne bi bilo zanesljivo. Način, kako upravljamo z napravami, je opisan za vsako napravo posebej, držali pa smo se smernic dobre uporabniške izkušnje in implementirali način, ki bo uporabniku kar se da naraven in preprost za uporabo.

Obstaja več načinov implementacije interakcije z objektom v 3D prostoru. V naši knjižnici smo implementirali interakcijo s kamero. Torej uporabnik vrti in premika kamero okrog določene točke (ponavadi center prizora ali objekta). Drugi dve možnosti, ki bi tudi prišli v upoštev v našem primeru pa sta interakcija z objektom, ki je primerna v primeru, kadar imamo samo en objekt, ali pa interakcija s celotnim prostorom, kjer je kamera statična in vrtimo ter premikamo prostor, v katerem se nahajajo objekti.

4.1 Miška

4.1.1 Upravljanje

Pri upravljanju z miško je način interakcije s kamero odvisen od tega, kateri gumb uporabnik pritisne. Na levi klik je vezana rotacija kamere, na desni klik pa premikanje kamere. Približujemo ali oddaljujemo (angl. *Zoom*) kamero s kolescem ali temu ekvivaletno gesto na drsni ploščici, če uporabljamo prenosnik. Za premikanje ali rotacijo kamere je potrebno držati ustrezni gumb in premikati miško.



Slika 4.1: Upravljanje z miško

4.1.2 Implementacija

Vsi sodobni brskalniki podpirajo implementacijo poslušalcev dogodgov (angl. *event listener*) za miško s pomočjo JavaScripta. Te dogodke brskalnik sproži ob vsakem premiku miške, pritisku gumba na miški ali vrtenju koleščka. Da

lahko izvedemo neko kodo, ko se zgodi en tak dogodek, moramo registrirati poslušalec. Primer ukaza, s katerim registriramo poslušalec na globalni objekt *window*:

```
window.addEventListener('mousedown', onMouseDown, false);
```

V naši knjižnici smo implementirali šest poslušalcev:

- *mousedown* - Ko uporabnik pritisne na enega izmed gumbov, začnemo s sledenjem premikov. Poslušalec registriamo ob inicializaciji.
- *mouseup* - Ko uporabnik spusti gumb, prenehamo s sledenjem in deregistriramo poslušalec. Registriramo ga, ko se zgodi dogodek *mousedown*.
- *mousemove* - Ko se zgodi *mousedown*, registriramo ta poslušalec in začnemo s sledenjem. Ko se zgodi *mouseup*, ga deregistriramo.
- *wheel* - Ko uporabnik zavrti koleščk na miški, sprožimo funkcijo, ki približa kamero. Registriramo ga ob inicializaciji.
- *contextmenu* - s tem poslušalcem preprečimo privzeto delovanje desnega klika, odpiranje kontekstualnega menija. Registriramo ga ob inicializaciji.
- *mouseout* - Če uporabnik med sledenjem premakne miško izven okna brskalnika, prenehamo in odjavimo ustrezne poslušalce. Registriramo ga ob inizializaciji.

Potek interakcije za rotacijo kamere je v kodi sledeč - ob inicializaciji modula miške se registrira poslušalec *mousedown*. Ko uporabnik pritisne in drži levi gumb, se sproži funkcija, ki registrira poslušalce *mouseup*, *mousemove* in *mouseout* ter nastavi način interakcije na rotacijo.

Od tega trenutka dalje, dokler se ne sproži *mouseup* in odjava poslušalcev, se ob vsakem premiku miške sproži dogodek *mousemove*, ki nosi informacije o premiku v obliki JavaScript objekta. Ta objekt je kot prvi argument podan v našo funkcijo, ki se pokliče ob vsakem premiku. Podatke, ki jih vsebuje ta

objekt določa konzorcij W3C [3]. Za nas pa sta v tem primeru pomembna samo dva podatka in sicer *clientX* in *clientY*, ki nam povesta trenutno pozicijo miškinega kurzorja v okvirju brskalnika. Enote so piksli in so relativni na levi zgornji kot okna, kjer je iztočišče koordinatnega sistema z vrednostjo *(0,0)*.

Ko dobimo ta dva podatka, lahko izračunamo razliko glede na prejšnjo pozicijo in na podlagi tega izračunamo, za kolikšno vrednost bo potrebno obrniti kamero. Ta dva podatka pretvorimo v naš 2D vektor za lažje računanje. Potem na podlagi načina interakcije, ki je trenutno aktiven, pokličemo ustrezno funkcijo, ki izračuna in nastavi novo pozicijo kamere. V tem primeru je to funkcija *rotate()*, ki izgleda tako:

```
function rotate () {
  scope.sphericalDelta.rotateLeft(
    2 * Math.PI * scope.mousePos.delta.x /
    scope.container.clientWidth * scope.rotateSpeed
  );
  scope.sphericalDelta.rotateUp(
    2 * Math.PI * scope.mousePos.delta.y /
    scope.container.clientWidth * scope.rotateSpeed
  );
}
```

Izsek kode 4.1: Primer izračuna rotacije kamere

V tej funkciji izracunamo delto kotov theta (levo/desno) in phi (gor/dol) za rotacijo sfernega koordinatnega sistema kamere. Kot izračunamo na podlagi delte premika, širine okna brskalnika in modifikatorja hitrosti rotacije, ki ga, kot že opisano, nastavimo v nastavitvah knjižnice.

Ko smo opravili vse potrebne izračune, sprožimo naš lastni dogodek, da je potrebna osvežitev prizora. Ta dogodek ni standardni, ker smo ga ustvarili po meri in ga sprožimo vsakič, ko spremenimo pozicijo kamere in tako sporočimo modulu za izris prizora, da mora upoštevati spremembe. Dogodek sprožimo

z ukazom:

```
window.dispatchEvent(scope.events.updateView(scope.INPUT.get()));
```

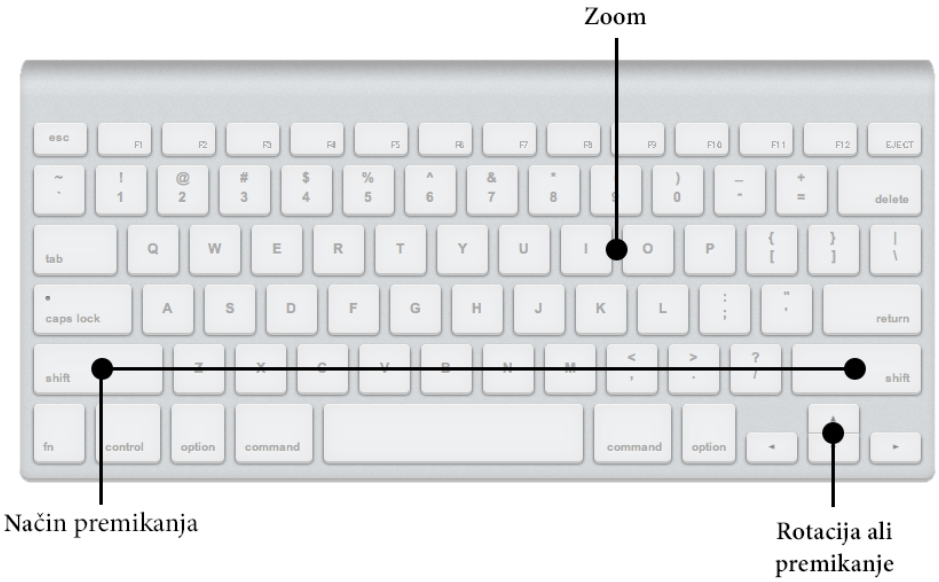
V dogodku pošljemo tudi podatek o trenutno uporabljeni vhodni napravi, kar uporabimo pri osveževanju stanja v HUD vrstici. Za tem trenutno pozicijo miške nastavimo kot izhodno točko za izračun pri naslednjem premiku. Ta postopek ponavljamo, dokler uporabnik premika miško in ne spusti gumba.

Ko uporabnik spusti gumb, deregistriramo vse tri poslušalce in nastavimo trenutni način interakcije in trenutno vhodno napravo na prazno vrednost *NONE*.

4.2 Tipkovnica

4.2.1 Upravljanje

Pri upravljanju s tipkovnico seveda uporabljamo tipke za interakcijo s kamero. Za rotacijo kamere uporabljamo smerne tipke (tipke s puščicami) in enako za premikanje, vendar moramo v drugem primeru istočasno držati še modifikator tipko Shift. Smerni tipki gor in dol sledita osi Y, tipki levo in desno pa osi X. Za približevanje ali oddaljevanje kamere uporabljamo tipki I in O (kratici za *in and out*). Za ponastavitev pozicije kamere na začetno stanje kliknemo na tipko R.



Slika 4.2: Upravljanje s tipkovnico

4.2.2 Implementacija

Tako kot dogodki za miško so tudi dogodki za tipkovnico podprti v vseh modernih brskalnikih. Za tipkovnico je definiranih manj tipov dogodkov kot za miško in sicer samo trije:

- *keydown* - Dogodek se sproži, ko uporabnik pritisne (navzdol) katerokoli tipko v standardnem naboru (razne bližnjice in multimedijske tipke v večini primerov niso podprte).
- *keyup* - Se sproži, ko uporabnik spusti tipko (navzgor). To je drugi del interakcije s tipkovnico.
- *keypress* - Se sproži na enak način kot *keydown* ob pritisku tipke navzdol, vendar samo v primerih, ko ta akcija sproži tudi izpis nekega znaka, ponavadi v vnosno polje. Uporaba tega dogodka se počasi opušča zavoljo uporabe dogodka *input* na vnosnih poljih.

Vsak dogodek, ki ga sproži tipkovnica, nosi tudi podatek o tem, katera tipka je bila pritisnjena. Vendar pa je zaradi različnih standardov skozi leta v večini modernih brskalnikov implemeniranih več različnih oblik te informacije. Na primer ob kliku smerne tipke gor se v najnovejšem spletnem brskalniku Chrome sproži dogodek, ki nosi med drugim tudi naslednje podatke:

```
KeyboardEvent {
    charCode: 0,
    code: "ArrowUp",
    key: "ArrowUp",
    keyCode: 38,
    keyIdentifier: "Up",
    which: 38
}
```

Izsek kode 4.2: Primer različnih identifikatorjev tipke v keydown dogodku

V primeru, da gre za *keydown* dogodek, je najbolje uporabljati podatek `keyCode`. V primeru, če bi poslušali za dogodek *keypress*, pa bi morali preverjati podatek `charCode`, ker je prišlo do zapisa v vnosno polje. Zaradi tovrstne nekonsistentnosti med podobnimi dogodki so novejši brskalniki uvedli še podatek *which*, ki vedno vrne pravo vrednost, vendar pa še ni podprt v vseh brskalnikih in ga moramo torej uporabljati previdno.

V naši knjižnici poslušamo dva dogodka tipkovnice - *keydown* za začetek interkacije in *keyup* za konec. Ko uporabnik drži tipko, se zaporedoma prožijo *keydown* dogodki, vendar je po prvem opazen krajši zamik. To funkcionalnost imajo implementirano vsi moderni brskalniki in je namenjena preprečevanju nanamernega držanja tipke. To pa ima slabo lastnost, saj v našem primeru, če uporabnik drži tipko, opazimo preskok po prvem premiku, kar pokvari uporabniško izkušnjo. Prav tako je težje implementirati podporo pritiska večih tipk hkrati, če se zanašamo na *keydown* dogodek.

Zato smo interakcijo s tipkovnico implementirali s pomočjo timerja oz.

časovnika. Ob vsakem *keydown* dogodku si zapomnimo, katera tipka je bila pritisnjena in dokler uporabnik ne spusti tipke, periodično izvajamo akcijo, ki jo ta tipka predstavlja. Ob *keyup* dogodku v objektu, ki hrani podatke o tem, katere tipke so bile pritisnjene, najdemo ustrezni vnos in ga pobrišemo ter pokličemo ustrezno ponastavitveno funkcijo. Ob pritisku na tipko preverimo tudi, ali smo pritisnili katero izmed meta tipk (CMD, Shift, Ctrl) ali tipko F5 in ne naredimo ničesar. To smo dodali zaradi lažjega razvoja. Poslušalce dogodkov se nastavi enako kot pri miški zato tega tu ne bomo posebej opisovali. Izsek iz kode si lahko ogledamo spodaj.

```
1  function onKeyDown (event) {
2    // Ne naredimo nič, če so pritisnjene tipke CMD, CTRL,
       Shift ali F5
3    if (event.metaKey ||
4        event.ctrlKey ||
5        pressed[event.keyCode] !== undefined ||
6        event.keyCode === keys.SHIFT ||
7        event.keyCode === keys.F5) return;
8
9    // Preprečimo privzeto delovanje tipk
10   event.preventDefault();
11
12   // Ob pritisku tipke R, ponastavimo pogled na privotno
       stanje
13   if (event.keyCode === keys.R) {
14     scope.win.dispatchEvent(scope.events.resetView);
15     return;
16   }
17
18   // Nastavimo tipkovnico kot trenutno vhodno napravo
19   scope.INPUT.set(scope.INPUTLIST.KEYBOARD);
20
21   // Nastavimo način interakcije na podlagi pritisnjene tipke
       in ali je bila hkrati pritisnjena tudi tipka Shift, za
       premik kamere
22   if (arrowKeys.indexOf(event.keyCode) > -1) {
```

```
23      if (event.shiftKey) {
24        scope.MODE.set(scope.MODELIST.PAN);
25      } else {
26        scope.MODE.set(scope.MODELIST.ROTATE);
27      }
28    } else if (zoomKeys.indexOf(event.keyCode) > -1) {
29      scope.MODE.set(scope.MODELIST.ZOOM);
30    } else {
31      // Če ni bila pritisnjena nobena od prednastavljenih tipk
            , ne nadaljujemo z izvajanjem
32      return;
33    }
34
35    // Tipko dodamo v objekt, ki hrani stanje pritisnjenih tipk
36    pressed[event.keyCode] = true;
37
38    switch (scope.MODE.get()) {
39      case scope.MODELIST.ROTATE:
40        switch (event.keyCode) {
41          case keys.LEFT:
42            // Nastavimo rotacijo
43            rotateX(1);
44            // Nastavimo ponastavitveno funkcijo
45            pressed[event.keyCode] = rotateX;
46            break;
47          // ...
48          // Preostala koda, ki nastavi ustrezno akcijo, ki pa
                je predolga, da bi jo izpisali v celoti
49    }
50
51    // Če še ne izvajamo osveževanja kamere, ga poženemo, druga
        če ne naredimo nič
52    if (!updating) {
53      // Nastavimo timer, ki bo izvajal osveževanje ob periodi,
            ki je nastavljena v glavnem modulu
54      timer = setInterval(update, scope.timerInterval);
55      updating = true;
56    }
```

```
57
58 }
59
60 // Ob keyup dogodku, najdemo vnos v našem objektu tipk, pokli
       čemo funkcijo, ki ponastavi interakcijo in pobrišemo tipko
61 function onKeyUp (event) {
62   if (pressed[event.keyCode] !== undefined) {
63     pressed[event.keyCode].call(null, 0);
64     delete pressed[event.keyCode];
65   }
66 }
67
68 // Funkcija, ki osvežuje pozicijo kamere
69 function update () {
70   // Če je objekt pritisnjenih tipk prazen, potem prenehamo z
         osveževanjem
71   if (!Object.keys(pressed).length) {
72     updating = false;
73     clearInterval(timer);
74     return;
75   }
76
77   // Zahtevamo, da se pred naslednjim izrisom na zaslon osvež
        i tudi pozicija kamere
78   requestAnimationFrame(function () {
79     scope.win.dispatchEvent(scope.events.updateView(
          scope.INPUT.get()));
80   });
81 }
```

Izsek kode 4.3: Primer *keydown* in *keyup* funkcij v modulu tipkovnice

4.3 3D miška

4.3.1 Upravljanje

3D miška 3Dconnexion SpaceNavigator je zmožna zaznavati premike na šestih oseh. Za rotacijo kamere analogni krmilni pokrovček miške nagibamo v levo/desno smer za rotacijo po Y osi, ter naprej/nazaj za rotacijo po X osi. Za premikanje kamere po X osi krmilnik potiskamo levo ali desno (brez nagibanja), za premikanje po Y osi pa krmilnik povlečemo navzgor oz. potisnemo navzdol. Za približevanje in oddaljevanje kamere zasukamo krmilnik v smeri urinega kazalca ali obratno. Za ponastavitev pozicije pritisnemo gumb na desni strani miške.

4.3.2 Implementacija

3D miška je nestandardna vhodna naprava in zato (v času pisanja) ni direktno podprta v nobenem spletnem brskalniku in zato nima svojih dogodkov, ki bi jih lahko poslušali, tako kot navadna miška in tipkovnica. Vendar pa je v večini novejših brskalnikov [4] podprt JavaScript vmesnik *GamePad API*, ki je primarno namenjen podpori igralnih ploščkov v spletnih igrah, vendar ga lahko v našem primeru uporabimo tudi za branje vrednosti osi naše 3D miške. Principa delovanja igralnih ploščkov in naše 3D miške sta zelo podobna. Obe napravi imata nekaj gumbov in enega ali več analognih krmilnikov. Največja razlika je ravno v slednjih, saj krmilniki na igralnih ploščkih zaznavajo premike v dveh smereh, 3D miška 3Dconnexion SpaceNavigator je zmožna zaznavati premike na šestih oseh. Vmesnik *GamePad API* ima to lastnost, da ob zagonu brskalnika ne zaznava priključenih igralnih ploščkov, ampak se preverjanje stanja začne šele po tem, ko uporabnik prvič pritisne enega izmed gumbov na ploščku.

```
1 // Periodično vsako sekundo preverjamo ali obstaja vsaj en
    plošček, ki je priključen. Nato začnemo z branjem
    vrednosti.
```

```
2 checkTimer = setInterval(function () {
3   if (navigator.getGamepads()[0] !== undefined && !connected)
        {
4     timer = setInterval(update, scope.timerInterval);
5     connected = true;
6     scope.INPUT.set(scope.INPUTLIST.GAMEPAD);
7   }
8 }, 1000);
```

Izsek kode 4.4: Primer preverjanja ali je naprava priklopljena

Implementacija skozi vmesnik *GamePad API* je relativno preprosta in sicer *navigator* objekt, ki je del globalnega objekta *window*, nosi informacijo o vseh trenutno aktivnih igralnih ploščkih oz. v našem primeru 3D miške in trenutne vrednosti vseh osi analognih krmilnikov ter gumbov. Te vrednosti lahko torej periodično preverjamo in na podlagi njih sprožimo ustrezno akcijo. Vrednosti osi so shranjene v urejenem seznamu, katerega vrstni red ni določen s specifikacijo vmesnika in je odvisen od modela priklopljene naprave. Ob branju se torej lahko zanesemo na vrstni red samo, če vedno priklopimo isto napravo in smo predhodno s testiranjem določili vrstni red branja. Vrednosti so zapisane v tipu *double* [5], torej so natančne na 17 decimalk. Razpon vrednosti je ponavadi od -1 do 1 (0 je v privzetem stanju), vendar pa je v primeru naše miške ta razpon malo manjši (najverjetneje posledica implementacije gonilnikov) in sicer od -0,38 do 0,38. Dodali smo tudi minimalno vrednost, ki jo mora vrednost neke osi preseči, da upoštevamo akcijo, saj je miška izredno občutljiva in natančna, kar nam izboljša uporabniško izkušnjo. V našem primeru se je po testih najbolje izkazala vrednost 0,01, saj ignorira samo najmanjše nenamerne premike. Seveda pa lahko to vrednost po želji uporabnik spremeni.

```
1 // Funkcija, ki se pokliče vsakih 30 ms in preverja vrednosti
      osi
2 var update = function () {
3   // Shranimo trenutno stanje za lažje sklicevanje
```

```
4     gamepad = navigator.getGamepads()[0];
5
6     // Če plošček ni povezan prenehamo z izvajanjem
7     if (gamepad === undefined) {
8       clearInterval(timer);
9       connected = false;
10      return
11    }
12
13    // Preverjanje vrednosti osi 5, ki sproži rotacijo kamere
          po X osi
14    if (Math.abs(gamepad.axes[4]) > scope.gamepadThreshold) {
15      rotateX(gamepad.axes[4] * scope.gamepadSensitivity);
16      hasChanged = true;
17    }
18
19    // Če je prišlo do spremembe, pred naslednjim izrisom na
          zaslon, osvežimo pozicijo kamere
20    if (hasChanged) {
21      requestAnimationFrame(function () {
22        scope.win.dispatchEvent(scope.events.updateView(
              scope.INPUT.get()));
23      });
24      hasChanged = false;
25    }
26  };
```

Izsek kode 4.5: Primer periodičnega branja vrednosti osi miške

4.4 Vmesnik Leap Motion

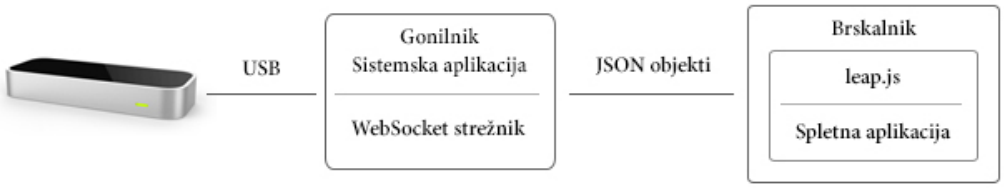
4.4.1 Upravljanje

Vmesnik Leap Motion je zmožen zaznavati, ali ima uporabnik iztegnjene dlani ali zaprte v pest. To funkcionalnost smo uporabili za zaznavanje začetka

interakcije. Ko uporabnik postavi iztegnjene dlani v zaznavno območje, jih lahko premika brez vpliva na kamero. Ko uporabnik dlan sklene v pest, se začne sledenje poziciji dlani. To uporabniku omogoča "preprijemanje"kamere in robustnejšo interakcijo. Z vmesnikom Leap Motion kamero vrtimo s pomočjo leve roke. Ko levo dlan sklenemo v pest jo lahko premikamo levo/desno za rotacijo okoli Y osi, ter gor/dol za rotacijo okoli X osi. Za premikanje kamere se uporablja desna roka in enaki premiki kot za rotacijo. Približevanje in oddaljevanje kamere lahko upravljamo z uporabo obeh rok naenkrat. Kamero približujemo tako, da pesti premikamo narazen, oddaljujemo pa tako, da jih potisnemo skupaj.

4.4.2 Implementacija

Tako kot 3D miška tudi vmesnik Leap Motion ni podprt v nobenem spletnem brskalniku, vendar z razliko, da zanj ni implementiran noben privzeti vmesnik API v JavaScriptu, ki bi z njim lahko direktno komuniciral. Zato so za delovanje naprave Leap Motion v spletnem brskalniku potrebni trije sestavni deli - sistemski gonilnik, sistemska aplikacija, ki zna komunicirati z napravo, izvaja izračune z zajetimi podatki in služi kot lokalni strežnik ter JavaScript knjižnica *leap.js* v spletnem brskalniku, ki se s pomočjo tehnologije *WebSocket* poveže na lokalni strežnik, s katerega nato prejema podatke [6].



Slika 4.3: Prikaz arhitekture WebSocket povezave z Leap Motion

Ko smo zagotovili vse tri predpostavke za delovanje, nam je v spletnem

brskalniku na voljo nov prototip *Leap.Controller()*, ki hrani podatke in nudi funkcije, ki nam olajšajo delo s podatki. Podatki so shranjeni v objektih za vsako sliko posebej, ki zajame naprava. Tipično lahko naprava zajema do 200 slik na sekundo, v spletnem brskalniku pa lahko dostopamo do zadnjih 200 slik (angl. *frames*) [7], kar pomeni, da imamo dostop do podatkov za najmanj 1 sekundo nazaj (odvisno od hitrosti zajema slik). Znotraj objekta vsake slike najdemo podatke o poziciji dlani, prstov, katere geste (angl. *gesture*) je uporabnik sprožil, itd. Skrajšan primer objekta ene slike lahko vidimo spodaj. Celotni objekt obsega še veliko več podatkov in zaradi spreminjanja vmesnika API skozi verzije vsebuje tudi veliko podvojenih podatkov pod drugačnimi imeni z namenom podprtja vseh verzij odjemalcev.

```
{
  currentFrameRate: // decimalna št., trenutna hitrost zajema slik na sekundo
  id: // decimalna št., unikatna id. številka slike
  timestamp: // cela št., časovna oznaka

  gestures: [{ // seznam objektov gest
    direction: // seznam decimalnih št. (3D vektor), smer geste
    duration: // cela št., trajanje geste v mikrosekundah
    handIds: // seznam id. številk rok uporabljenih v gesti
    id: // cela št., unikatna id. številka geste
    fingerIds: // seznam id. številk prstov
    position: // seznam decimalnih št. (3D vektor), pozicija roke
    type: // beseda, tip geste
  }, ...]

  hands: [{ // seznam objektov rok
     confidence: // decimalna št., zanesljivost podatkov
     direction: // seznam decimalna št. (3D vektor), smer v katero je usmerjena roka
     grabStrength: // decimalna št., moč stiska roke
```

```
20      id: // cela št., id. številka roke
21      palmPosition: // seznam decimalnih št. (3D vektor),
            pozicija centra dlani
22      type: // beseda, leva ali desna roka
23    }, ...]
24
25    fingers: [{ // seznam objektov prstov
26      bases: // seznam vektorjev za vsako kost v prstu
27      direction: // seznam decimalnih št. (3D vektor), smer
            prsta
28      extended: // logična vrednost (true ali false), ali je
            prst iztegnjen
29      handId: // cela št., id. številka roke kateri pripada
30      id: // cela št., id. številka prsta
31      length: // decimalna št., dolžina prsta
32      type: // cela št., zaporedna št. prsta na roki: 0 -
            palec, 4 - mazinec
33    }, ...]
34  }
```

Izsek kode 4.6: Primer objekta dlani in opis nekaterih podatkov

Preden lahko začnemo brati podatke z naprave, moramo ustvariti nov primerek prej omenjenga prototipa in se povezati na strežnik. Potem lahko nastavimo poslušalce na naš primerek, ki se bodo sprožili, ko naprava prične ali zaključi z oddajanem podatkov.

```
1  // Ustvarimo nov primerek prototipa
2  this.leap = new Leap.Controller();
3  this.leap.connect();
4
5  // Povežemo poslušalce za začetek in konec oddajanja
6  this.leap.on('streamingStarted', startUpdating);
7  this.leap.on('streamingStopped', stopUpdating);
8
9  // Začnemo periodično brati podatke
10 function startUpdating () {
```

```
11    timer = setInterval(update, scope.timerInterval);
12  }
13
14  // Ustavimo branje
15  function stopUpdating () {
16    clearInterval(timer);
17  }
```

Izsek kode 4.7: Inicializacija povezave z vmesnikom

Ob vsakem klicu funkcije *update* si najprej shranimo trenutno stanje in stanje predhodne slike, da lahko izračunamo razliko v poziciji dlani in ustrezno premaknemo kamero. Preden lahko nadaljujemo z izračuni, moramo preveriti, ali obe, trenutna in predhodnja slika sploh vsebujeta roke, saj je po umiku rok izven zaznavnega območja naprave, urejeni seznam, ki hrani podatke o dlaneh, prazen. Nato preverimo, katera dlan je trenutno v vidnem polju in ali je palec na njej iztegnjen ali pokrčen in si te podatke shranimo. Vsak objekt dlani hrani tudi podatek o zanesljivosti podatkov o poziciji in prstov (decimalna vrednost med 0 in 1), ki jo preverjamo zavoljo boljše uporabniške izkušnje. Da bomo upoštevali premik, mora biti zanesljivost podatkov nad 0,5.

S temi podatki lahko nato izvedemo ustrezno akcijo. V vsakem objektu slike je seznam objektov rok, ki hrani seznam objektov za vsak prst na tej roki, in podatek, ali je iztegnjen, v obliki *Boolean* vrednosti. Objekt prsta hrani tudi 3D vektorje za pozicijo vsake kosti v prstu, s katerimi lahko potem izračunavamo bolj kompleksne interakcije, kot so na primer prijemanje objektov v 3D prostoru in podobno. Poleg prstov vsak objekt roke hrani tudi podatek o poziciji dlani, kar uporabljamo tudi mi v naši implementaciji. Ta podatek je sicer manj natančen, saj je izračunan na podlagi približka centra dlani. V primerih, ko so prsti roke pokrčeni, je težko izračunati center dlani, saj jo prsti zakrivajo, vendar je za naše potrebe podatek dovolj natančen in med testiranjem nismo opazili velikih napak.

```
1  // Shranimo trenutno in predhodnjo sliko
2  curr = scope.leap.frame(0);
3  prev = scope.leap.frame(1);
4
5  // Preverimo, če je v obeh slikah prisotna vsaj ena roka
6  if (curr.hands.length && prev.hands.length) {
7    scope.INPUT.set(scope.INPUTLIST.LEAP);
8
9    // Sprehodimo se skozi seznam rok
10   for (i = 0, len = curr.hands.length; i < len; i++) {
11
12     // Preverimo če je zanesljivost podatkov dovolj visoka
13     if (curr.hands[i].confidence > scope.leapThreshold) {
14
15       // Shranimo vrstni red rok kot identifikator, ter ali
             je palec iztegnjen
16       if (curr.hands[i].type === 'left') {
17         left.id = i;
18         left.tracking = !curr.hands[i].fingers[
               scope.leapTriggerFinger].extended;
19       } else {
20         right.id = i;
21         right.tracking = !curr.hands[i].fingers[
               scope.leapTriggerFinger].extended;
22       }
23     }
24   }
25
26   // Primer kode, kjer preverimo ali je trenutno aktivna leva
         roka
27   if (left.tracking) {
28
29     // Z levo roko vrtimo kamero
30     scope.MODE.set(scope.MODELIST.ROTATE);
31
32     // Preverimo ali v predhodnji sliki tudi obstaja ta roka
33     if (prev.hands[left.id]) {
34
```

```
35        // Izračunamo razliko v pozicijah dlani in kličemo
              rotacijsko funkcijo
36        rotate(getDelta(curr.hands[left.id].palmPosition,
              prev.hands[left.id].palmPosition));
37      }
38    }
39  }
```

Izsek kode 4.8: Potek izračuna pozicije dlani

Poglavje 5

Izris slike

V tem poglavju bomo na kratko opisali, kako izrišemo naš 3D prizor na zaslon, čeprav to ni bil cilj diplomskega dela.

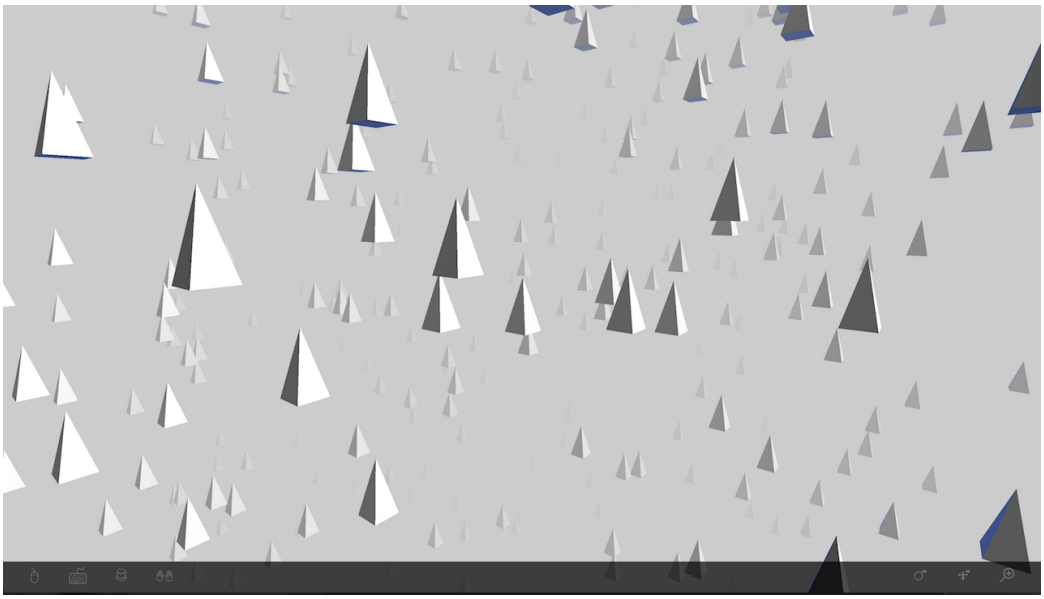
5.1 WebGL in knjižnica three.js

WebGL je vmesnik API v JavaScriptu, s pomočjo katerega lahko izrisujemo interaktivno 3D in 2D grafiko v podprtih brskalnikih. Je standardiziran po specifikaciji kozorcija Khronos in je popolnoma integriran v brskalnik, kar omogoča uporabo grafične procesne enote (angl. *GPU*) za izračun fizikalnih enačb in procesiranja slik. WebGL je izpeljanka grafičnega API vmesnika OpenGL ES 2.0 in za izrisovanje slik v spletnem brskalniku uporablja HTML element *canvas*. Avtor WebGL-a je srbski programer Vladimir Vukićević, ki je bil v času razvoja zaposlen pri Mozilli. Prva standardizirana verzija vmesnika je bila izdana leta 2011, trenutno pa je v pripravi verzija 2.0, ki bo temeljila na vmesniku OpenGL ES 3.0.

V naši knjižnici smo pa za izris slike uporabili visokonivojsko JavaScript knjižnico *three.js* (`https://threejs.org/`), ki zelo poenostavi uporabo WebGL-a skozi svoje prototipe in funkcije, poleg tega pa skrbi tudi za optimizacijo hitrosti izrisa slik in animacij.

5.2 Priprava in izris prizora

V naši implementaciji prizora smo uporabili enega izmed preprostih primerov uporabe, ki jih najdemo na spletnem mestu knjižnice *three.js* in ga priredili našim potrebam. Prizor obsega prazen 3D prostor, v sredini katerega je na naključnih mestih postavljenih 500 piramid ter tri luči, eno ambientno in dve usmerjeni. Dodana je tudi megla, ki ustvari občutek globine.



Slika 5.1: Prikaz izgleda prizora

Datoteka *interact-scene.io* je sestavljena iz dveh večjih delov kode - prvi del skrbi za začetno pripravo in nato za periodični izris prizora, ter drugi, ki izračunava pozicijo kamere na podlagi podatkov prejetih iz ostalih modulov.

Ob inicializaciji modula prizora se pokliče funkcija, ki ustvari nov primerek prizora in jo shrani v globalni doseg knjižnice (angl. *global scope*), zato, da nam je na voljo skozi konzolo brskalnika tudi po koncu inicializacije. Nato v prizor dodamo meglo, ki je sive barve in 99,8% prosojna ter ustvarimo nov primerek *WebGLRenderer* prototipa, ki je najpomembnejši

del kode, saj skrbi za komunikacijo in pretvarjanje *three.js* ukazov v obliko, ki so razumljivi vmesniku WebGL, ki nato izriše sliko v podani element *canvas*. Nastaviti mu moramo tudi barvo ozadja, razmerje pikslov naprave (1.0 za navadne naprave in večje, če gre za naprave z zelo visoko resolucijo, ki uporabljajo več fizičnih pikslov za izris enega na sliki) ter velikost elementa, kamor izrisujemo v pikslih. V našem primeru je to kar celotna površina brskalnika. Na koncu lahko element, v katerega bomo izrisovali, dodamo v DOM strukturo strani.

```
1  // Ustvarimo nov primerek prizora in dodamo meglo
2  scope.scene = new THREE.Scene();
3  scope.scene.fog = new THREE.FogExp2(0xcccccc, 0.002);
4
5  // Ustvarimo nov primerek WebGLRenderer prototipa in mu
       podamo nastavitve
6  scope.renderer = new THREE.WebGLRenderer();
7  scope.renderer.setClearColor(scope.scene.fog.color);
8  scope.renderer.setPixelRatio(window.devicePixelRatio);
9  scope.renderer.setSize(window.innerWidth, window.innerHeight)
       ;
10
11 // Dodamo <canvas> element v DOM strukturo
12 scope.container.appendChild(scope.renderer.domElement);
```

Izsek kode 5.1: Primer inicializacije prizora

Naslednji korak je priprava kamere in luči, ter postavitev piramid v prostor. Uporabljamo perspektivno kamero, *three.js* pa ima na voljo tudi ortografsko kamero. Ob inicializaciji ji moramo podati zorno polje, razmerje robov slike kamere in bližino ter daljino rezalne ravnine (definira, katere objekte vidimo glede na oddaljenost od kamere) in jo nato postavimo v sredino našega prostora. Sledijo piramide, za katere najprej določimo obliko in material, ki jih bo pokrival. Na podlagi tega ustvarimo 500 primerkov in jih postavimo na naključne X, Y in Z koordinate v prostoru. Nastavimo jim, da

so statičnega tipa in se ne bodo premikale v prostoru, kar omogoči WebGL-u optimizacijo izrisa in tako pohitrimo delovanje. Na koncu še dodamo vsako piramido v prostor. Na tej točki, čeprav smo v prostor dodali že vse elemente, bi se nam na zaslon izrisala črna slika, saj prostor še nima nobenega vira svetlobe, ki bi osvetlila piramide. Zato v prizor najprej dodamo eno ambientno luč, ki je temno sive barve, kar je enako kot če bi dodali zelo nežno belo svetlobo, nato sledita pa še dve usmerjeni luči, prva čisto bela in druga modra, ter ju postavimo na nasprotne strani prostora.

```
1  // Pripravimo kamero
2  scope.camera = new THREE.PerspectiveCamera(60,
       window.innerWidth / window.innerHeight, 1, 1000);
3
4  // Pripravimo obliko in material za piramide
5  var geometry = new THREE.CylinderGeometry(0, 10, 30, 4, 1);
6  var material =  new THREE.MeshPhongMaterial({color:0xffffff,
       shading: THREE.FlatShading});
7
8  // Ustvarimo 500 piramid in jih postavimo v prostor
9  for (var i = 0; i < 500; i ++) {
10   var mesh = new THREE.Mesh(geometry, material);
11   mesh.position.x = (Math.random() - 0.5) * 1000;
12   mesh.position.y = (Math.random() - 0.5) * 1000;
13   mesh.position.z = (Math.random() - 0.5) * 1000;
14   mesh.updateMatrix();
15   mesh.matrixAutoUpdate = false;
16   scope.scene.add(mesh);
17 }
18
19 // Dodamo luči
20 var light = new THREE.DirectionalLight(0xffffff);
21 light.position.set(1, 1, 1);
22 scope.scene.add(light);
23
24 light = new THREE.AmbientLight(0x222222);
```

```
25 scope.scene.add(light);
```

Izsek kode 5.2: Primer priprave kamere, luči in piramid

Preden začnemo z dejanskim izrisom, dodamo še par poslušalcev za spremembo velikosti okna, ki poskrbijo, da se izris prizora ustrezno prilagodi novi velikosti. Na koncu pa pokličemo rekurzivno funkcijo, ki začne z izrisovanjem in intervalno osvežuje sliko. Ta interval ni vnaprej določen tako kot tisti za zajem podatkov, ampak je odvisen od zmogljivosti računalnika, saj je vezan na hitrost osveževanja izrisa na zaslon. Idealno je ta interval 60 slik na sekundo, vendar je lahko pri zahtevnejših 3D objektih tudi nižji. S tem preprečimo izračun slik, ki se nikoli ne izrišejo in ne zakasnimo izrisa na zaslon zaradi izvajanja kode.

```
1 function render () {
2   // Pred naslednjim izrisom na zaslon, ponovno izvedemo to
        funkcijo
3   requestAnimationFrame(render);
4   scope.renderer.render(scope.scene, scope.camera);
5 }
```

Izsek kode 5.3: Rekurzivna funkcija, ki izrisuje sliko na zaslon

Drugi del izrisa pa sestavlja izračunavanje pozicije kamere. Ta se ne računa periodično, ampak samo takrat, ko pride do spremembe in mi sprožimo dogodek za osvežitev. Vsak izračun se začne z novim primerkom 3D vektorja, ki bo hranil nove koordinate in smer kamere. Nastavimo mu, s katero osjo mora biti kamera vedno poravnana, v našem primeru je to os Y, zato da je kamera ne glede na rotacije vedno obrnjena tako, da os Y kaže gor. Izračunamo in shranimo razdaljo med trenutno pozicijo kamere in našo tarčo, ki je na začetku središče 3D prostora, vendar se spremeni s premikom kamere. Nato premaknemo kamero v center koordinatnega sistema (0,0,0), izračunamo nove kote, pod katerimi mora biti kamera obrnjena in po potrebi

premaknemo kamero, če je uporabnik sprožil to akcijo. Ko računamo kote kamere, upoštevamo tudi maksimalne kote za Y os, ki nam preprečujejo, da gremo čez vertikalo in posledično kamero obrnemo na glavo. Izračunamo tudi nov radij kamere, če je uporabnik spožil akcijo za približevanje kamere. Nato kamero premaknemo na pozicijo tarče in prištejemo nazaj razdaljo, ki smo jo shranili in tako dobimo novo pozicijo kamere. Celotna koda obsega več kot 120 vrstic, zato je na tej točki ne bomo navajali.

Poglavje 6

Sklepne ugotovitve

V tem diplomskem delu smo prikazali implementacijo JavaScript knjižnice za navigacijo v navideznem 3D prostoru znotraj spletnega brskalnika z uporabo dveh standardnih in dveh alternativnih vhodnih naprav. Miška in tipkovnica sta povsem standardni vhodni napravi za vsakega uporabnika računalnikov in tudi njuna implementacija je standardno podprta v vsakem spletnem brskalniku, medtem ko sta 3D miška in vmesnik Leap Motion nestandardni napravi in posledično nista podprti neposredno. Z uporabo JavaScript vmesnika *WebGL API* smo izdelali tudi preprost primer 3D prizora, v katerem z našo knjižnico upravljamo kamero.

Končni izdelek je JavaScript knjižnica, ki je razdeljena na sedem modulov - en glavni, ki definira prototip knjižnice v katerem se hranijo podatki in osnovne funkcionalnosti, ter šest modulov, ki razširjajo glavni prototip, od tega štirje za vsako vhodno napravo, en modul za izris 3D prizora in en za izris statusne vrstice. Vsi moduli imajo poleg zasebnih podatkov in funkcij dostop tudi do vsega v glavnem modulu.

Najzahtevnejša je bila implementacija 3D miške in vmesnika Leap Motion, ki nista direktno podprta v spletnih brskalnikih. V primeru 3D miške smo morali najprej najti način, kako sploh dostopati do podatkov. To je v novejših spletnih brskalnikih mogoče z uporabo vmesnika *Gamepad API*, ki je bil skupaj z *WebGL*-om predstavljen v specifikaciji *HTML5*. Primarno je le

ta namenjen povezovanju z igralnimi ploščki, vendar 3D miška uporablja enak princip analognih osi, zato je branje vrednosti relativno preprosto. Vmesnik Leap Motion pa je zahteval uporabo dodatne JavaScript knjižnice, ki preko tehnologije *WebSocket* komunicira s sistemsko aplikacijo in tako omogoči dostop do podatkov. Ti so podani v objektu *JSON* specifične strukture in je za razumevanje potrebno prebrati dokumentacijo na spletnem mestu proizvajalca naprave. Obe napravi zahtevata tudi nekaj privajanja na upravljanje, saj sta obe izjemno občutljivi. Še posebej vmesnik Leap Motion je nekaj čisto posebnega, ker uporabniku ne nudi nobene fizične povratne informacije. Način interakcije je popolnoma drugačen od ostalih naprav, a hkrati bolj naraven kot naprimer navadna tipkovnica, saj je premikanje roke podobno premikanju dejanske kamere.

Drugi največji izziv pa je bil, kako sploh strukturirati samo knjižnico, saj smo želeli, da je modularna, lahko razširljiva in uporabna tudi za ostale. Za vzgled smo vzeli popularne JavaScript knjižnice, kot naprimer *jQuery*. Objavili smo jo na spletnem mestu Github, kjer je dostopna vsem in morda bo komu služila kot vzorec za implementacijo 3D miške ali vmesnika Leap Motion s spletno 3D grafiko. Knjižnica je že uporabljena v projektu Med3D [11], ki so ga razvili na Fakulteti za računalništvo in informatiko.

Kljub temu, da smo stremeli k modularnosti in neodvisnosti kode, smo morali za nemoteno preklapljanje med vmesniki implementirati sistem proženja lastnih dogodkov, kar lahko ob implementaciji dodatnih modulov povzroči zmedo in oteži skalabilnost knjižnice. Prav tako je oblika podatkov, ki jih pridobimo z vmesnikov, prilagojena za uporabo z našim primerom 3D prizora in *three.js* knjižnice. Ti dve pomanjkljivosti naše trenutne rešitve bomo poskušali rešiti v nadaljnjem delu na knjižnici.

Poleg popravkov pa želimo dodati tudi nove funkcionalnosti. Prva je možnost spreminjanja načina interakcije za posamezni vmesnik, npr. spreminjanje tipk na tipkovnici, osi za 3D miško ali geste za Leap Motion. Deloma je to že mogoče skozi spreminjanje spremenljivk ob inicializaciji modula, vendar bi bil grafični vmesnik veliko boljša rešitev. Želimo pa se podati tudi v

svet virtualne resničnosti (angl. *virutal reality* ali VR), ki je v zadnjem letu postala zelo priljubljena. Ko smo analizirali primere uporabe vmesnika Leap Motion, smo našli tudi primere uporabe skupaj z VR napravami, kar potem omogoča ne le zaznavanje premikanja uporabnika v 3D prostoru, ampak tudi simulacijo rok, ki jih zaznava Leap Motion, nameščen na VR očala.

Literatura

[1] ECMAScript versions. [Online]. Dosegljivo: https://en.wikipedia.org/wiki/ECMAScript#Versions. [Dostopano 14.8.2016].

[2] Free vector icons. [Online]. Dosegljivo: http://www.flaticon.com/. [Dostopano 14.8.2016].

[3] Mousemove event. [Online]. Dosegljivo: https://www.w3.org/TR/uievents/#event-type-mousemove. [Dostopano 14.9.2016].

[4] Can I use - Browser support table for Gamepad API. [Online]. Dosegljivo: http://caniuse.com/#feat=gamepad. [Dostopano 26.9.2016].

[5] W3C Gamepad API specification. [Online]. Dosegljivo: https://www.w3.org/TR/gamepad/#attributes. [Dostopano 28.9.2016].

[6] Leap Motion WebSocket Communication. [Online]. Dosegljivo: https://developer.leapmotion.com/documentation/v2/javascript/supplements/Leap_JSON.html. [Dostopano 28.9.2016].

[7] Leap Motion Frames. [Online]. Dosegljivo: https://developer.leapmotion.com/documentation/javascript/devguide/Leap_Frames.html. [Dostopano 28.9.2016].

[8] Learning JavaScript Design Patterns, Addy Osmani. [Online]. Dosegljivo: https://addyosmani.com/resources/essentialjsdesignpatterns/book/#prototypepatternjavascript. [Dostopano 3.9.2016].

[9] 3Dconnexion SpaceNavigator. [Online]. Dosegljivo: https://www.cs-software.com/hardware/3Dconnexion/spacenavigator.html. [Dostopano 8.9.2016].

[10] Motion-controlled Servos with Leap Motion & Raspberry Pi. [Online]. Dosegljivo: https://www.pubnub.com/blog/2015-08-19-motion-controlled-servos-with-leap-motic [Dostopano 8.9.2016].

[11] Med3D volumetric visualization. [Online]. Dosegljivo: http://lgm.fri.uni-lj.si/portfolio-view/med3d-volumetric-visualization/. [Dostopano 25.11.2016].